C/S 模型在实时操作系统设计中的应用与研究①

Application of C/S Model in Real-Time Operating System

叶新栋 唐志强 涂时亮 (复旦大学 计算机学院 上海 200433)

摘 要: 良好的操作系统模型设计是一款优秀操作系统的核心和基础。首先介绍了操作系统内核设计中几种不同的设计模型理念，并在此基础上着重对如何利用 C/S 模型的消息传递机制来简化操作系统的设计做了深入研究。通过对 QNX 等一系列实时内核的深入研究，分析并总结了如何利用 C/S 模型来解决实时内核设计的一系列关键技术，对实时操作系统的内核设计有一定的指导意义。

关键词: 操作系统 C/S 模型 死锁 同步机制 QNX

20 世纪 50 年代中期到后期开发的早期操作系统很少考虑到结构问题，没有人具有构造大型软件系统的经验，并且对于互相依赖和交互的问题考虑过低。因此，那些单体结构的操作系统往往代码的冗余度很高，并且移植性和可调试性都很差。层次化的设计结构的出现将模块化程序设计技术引入到操作系统的设计中，这大大提高了操作系统的性能。然而，层次化的结构设计往往存在着内核规模过大和实时性能不高等一些不足之处。基于消息传递机制的微内核设计方案有效地弥补了层次化设计方案的不足之处，非常适合那些对实时性能要求极高的嵌入式系统。

1 层次化和微内核模型简介

层次化模型是实现系统设计模块化的一种常用的方法，这种技术将操作系统分为若干个层，每一层构建在下面一层之上。最低层是硬件，最高层是用户接口。操作系统的一层是对一个抽象对象的实现，它封装了数据和对这些数据的操作。图 1 描述了一个典型的操作系统层。

它由数据结构和可由更高调用的一系列程序构成。N 层也能够调用更低层的操作。

层次化设计最大的优点是其模块化设计。通过模块化设计，上一层次的操作或服务只会调用更低一层次的服务，因此，它对层次间的信息起到了很好的隐

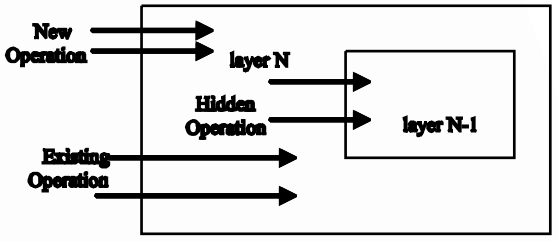

图 1 层次化模型中的层次图

藏作用。同时，当我们需要修改系统的相关服务时，并不需要修改整个系统的结构，只需要修改相关的层次便能达到我们的目的。然而，在实际系统的设计中我们很难准确的对众多系统的服务进行分层。层次化设计的最后一个问题是它的效率低于其它类型的系统。例如，当一个用户程序执行一个 I/O 操作时，它就要执行一个自陷到 I/O 层的系统调用，这个调用再调用存储器管理层，再由存储器管理层调用 CPU 调度层，然后 CPU 调度层转到硬件。在每一层中，可能要修改参数，可能要传递数据，等等。每一层都要增加系统调用的开销，最终要比采用非分层的系统花费更多的时间[1]。

微内核结构用一个水平分层的结构代替了传统的纵向分层的结构。在微内核外部的操作系统构建都被当做服务进程实现，他们可以借助于通过内核的消息传递机制来实现互相的交互。图 2 显示了当文件应用

① 收稿时间:2009-02-24

程序想要打开一个文件所执行的一系列操作。在微内核的结构设计中，只有那些具有最基本的操作系统的功能被放到内核中，这大大简化了实时内核的设计。同时，利用消息传递机制有效的避免了层层传递带来的时间开销，对系统的实时性和稳定性都有一定的提高。不仅如此，微内核还包括一致接口、可扩展性、灵活性、可移植性、可靠性、分布式系统支持以及对面向对象操作系统的支持等一系列优点[2]。

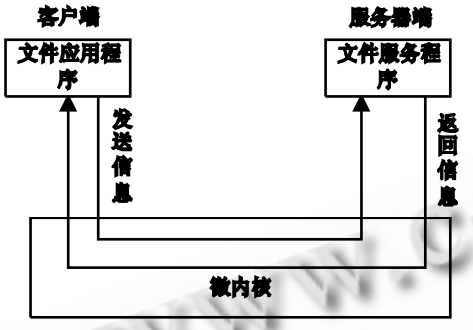

图 2　客户端和服务器端交互图

2　C/S模型在微内核中的架构

客户/服务器模型是微内核设计中的核心。任何应用程序想要得到系统的服务时，只需通过管道向内核发送消息，内核负责将消息传递给相应的服务程序，而服务程序接收到消息后，负责完成相应的操作，并最终将结果通过内核返回给应用程序。

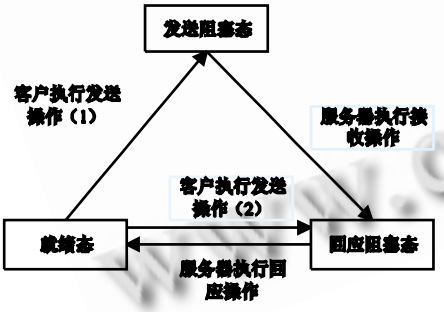

图 3　客户端状态机模型

如何设计客户端应用程序和服务器端服务程序的状态机模型是 C/S 模型机制的关键问题。通过分析 QNX 实时内核的 C/S 模型实现机制，我们提炼出了在客户端和服务器通用的两个状态机模型。图 3 显示的是客户端状态机模型，客户执行发送操作 1 代表的是当服务器处在就绪态时客户程序执行发送操作所发生的状态转换；客户执行发送操作 2 代表的是当服务器处在接收阻塞态时客户程序执行该操作所发生的状态转换。

图 4 显示的是服务端状态机模型，服务器执行接收操作 1 代表的是当客户端处在就绪态时服务端程序执行该操作时发生的状态转化；服务器执行接收操作 2 代表的是当客户端处在发送阻塞态时服务端程序执行该操作时发生的状态转化。

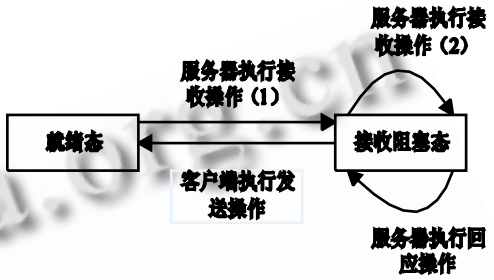

图 4　服务端状态机模型

3　关键问题的解决方案

C/S 模型的架构方案极大的简化了内核的设计，但同时由于 C/S 模型采用了消息传递机制来为各种服务传递信息，因此系统中大量的交互消息的存在给进程的同步、死锁的防止都带来了特殊的问题。同时，如何防止优先级逆转的发生也是 C/S 模型在操作系统内核设计中必需解决的关键问题。

3.1 自同步的实现

在 C/S 模型中，线程并不是直接相互通信。其必需遵循以下原则：若服务器端要接收某一消息，其必需先建立一专用通道；同时，若某一客户进程要向该服务进程发送一消息，其必需首先连接到服务器创建的通道上。图 5 显示了这一过程。

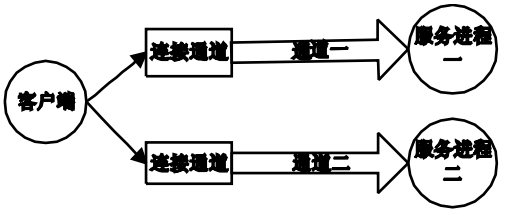

图 5　客户和服务端通道连接图

消息传递机制在 C/S 模型中不仅可以实现线程的通信，而且提供线程的同步。图六显示的是一个简单的自同步过程。在执行消息传递之前，服务进程 Process B 必须建立相关通道，而客户进程 Process A 必需连接到该通道上。

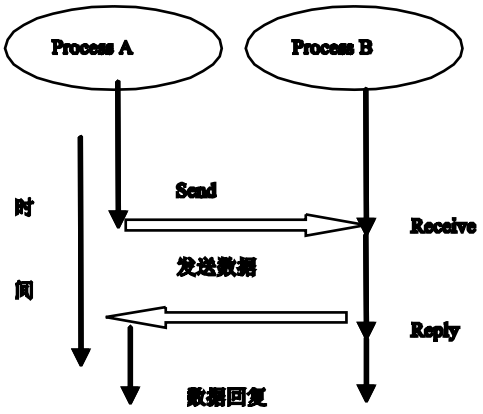

图 6 线程的自同步示意图[3]

3.2 优先级逆转的预防

优先级逆转指的是高优先级任务需要等待低优先级任务释放资源，而低优先级任务又在等待中等优先级任务的现象。这种现象会发生在进程竞争申请资源时，因此大部分 RTOS 都会在互斥锁中实现相关的协议来防止优先级逆转的发生。然而，采用 C/S 模型设计的微内核，发生优先级逆转的可能性却大得多。图七显示了基于C/S模型的微内核发生优先级逆转的一个简单模拟。图中序号代表了进程的优先级，操作 1 代表的是 Client1 发送消息给 Server，操作 2 代表的是 Client2 发送消息给 Server，操作 3 代表的是 Server 完成 Client2 的请求操作，操作 4 代表的是 Server 完成 Client1 的请求操作。

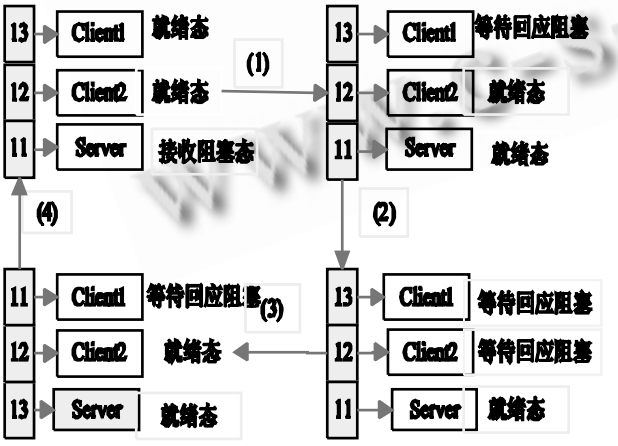

图 7 优先级逆转简单模拟图

从图 7 的分析可得，之所以在执行操作 2 后高优先级的进程 Client1 被低优先级的进程 Client2 所抢占，究其原因主要是：当客户进程申请服务进程执行相关操作后，客户进程的优先级就被与之关联的服务进程的优先级所取代，因而任何优先级高于该服务进程的其他客户进程都可以抢占执行。为了防止该类优先级逆转问题的发生，我们可以采取一种服务进程继承客户进程的原则来优化调度机制。该原则的核心是：当服务进程接收到某个客户进程的消息时，若其优先级低于该客户进程，则服务进程的优先级设置为客户进程同等大小。当服务进程释放操作后，其优先级自动恢复到原始状态。图 8 显示了经优化后的整个模拟过程，操作 1 和 2 与图 7 相同；而操作 3 代表的是 Server 完成 Client1 的请求操作，操作 4 代表的是 Server 完成 Client2 的请求操作。

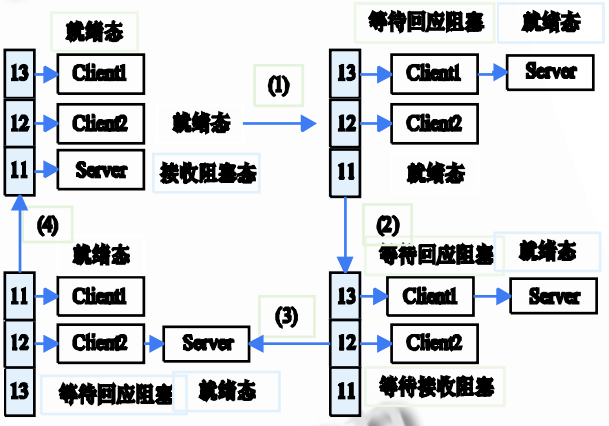

图 8 优化调度后的模拟图

3.3 死锁的避免

图 9 简单模拟了操作系统中死锁发生的整个过程，当进程 P1 拥有 P2 想要的资源 R1，同时 P2 拥有 P1 想要的资源 R2 时，死锁便发生了。资源申请及占有的无序化是导致死锁发生的一个主要原因，在 C/S 模型中，服务端进程亦可被当作一种资源，因而客户进程如何合理、有序地申请服务进程资源是解决死锁问题的关键。

在基于 C/S 模型的内核中，只要遵循两个原则便可以避免因竞争服务资源而引发的死锁问题。其一，任意两个进程不可同时向对方申请服务；其二，将进程申请服务的次序按树状结构排序，任意时刻只允许子进程向父进程申请服务(如图 10 所示)。例如，当某一客户进程想要查询相关数据，其首先因向数据库管理进程申请相关服务，再由数据库管理进程向文件管

理进程申请该服务。由于采取了有序的资源申请方式，因而任意两进程都不会因同时等待对方的服务而造成死锁的发生[4]。

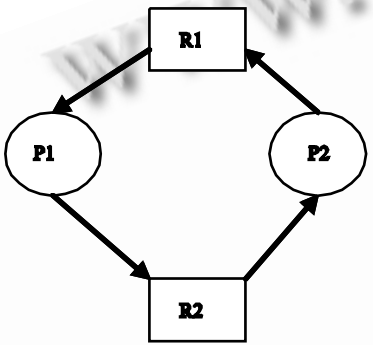

图 9　死锁模拟图

4　结论

采用 C/S 模型设计的实时内核能够有效的提高系统的实时性，同时针对由 C/S 模型带来的一系列新的挑战，如同步的实现、死锁的防止等问题，都能通过设计一种合理的方法予以解决，这大大提高了系统的稳定性。因此，采用 C/S 模型设计的实时内核在未来的操作系统设计中有着广阔的应用前景。

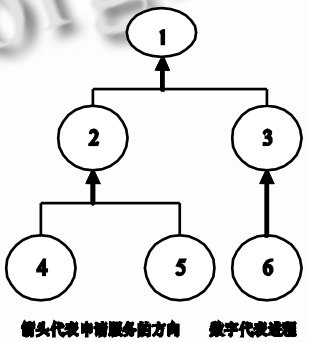

图 10　资源申请结构图

参考文献

1 Silberschatz A,et al. 郑扣根, 译. Operating System Concepts. Sixth Edition. 北京: 高等教育出版社, 2008.105－106.

2 Stallings W. 陈渝, 译. Operating Systems—Internals and Design Principles. Fifth Edition. 北京: 电子工业出版社, 2006.126－127.

3 QNX Real time Operating System —System Architecture (Third Edition). Canada: QNX Software Systems, 2005.

4 QNX Neutrino RTOS System Architecture. Canada: QNX Software Systems, 2008.